Amsoft DEVPAC

The Complete HiSoft Assembler, dis-assembler, editor and monitor. Revised and enhanced for the AMSTRAD CPC464

Published by AMSOFT, a division of
Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
Brentwood
Essex

SOFT116
Copyright © 1984 HiSoft and Amstrad Consumer Electronics plc

Foreword

HiSoft's DEVPAC has been widely recognised as the most complete software tool for low level machine code programming. It incorporates virtually every feature available on much larger microcomputer systems, and is the product of several years development and experience. Amsoft is particularly grateful to David Link for his efforts to enhance the program to take advantage of the advanced features of the CPC464.

The AMSOFT user group magazine will be continuing the general background to machine language programming with the CPC464; and the Concise Firmware manual (SOFT 158) is an essential companion publication for users wanting to gain access to the wealth of built-in firmware that binds the BASIC and hardware together in such an effective manner.

Page numbers are presented in the form:

(Section).(Page Number)

Updates and further information may be published from time to time in the Amstrad user club magazine.

Notes on style in this manual

Please note that the typestyles used in AMSOFT publications are intended to help identify the different operations and sequences used in computer operation:

Text that is typed at the keyboard and appears on the screen is shown in an OCR-B typeface:

```
10 FOR N= 1 TO 50
```

Where clarity is enhanced, keyboard actions that instruct a command sequence but do not necessarily have a corresponding representation on the screen are shown in Helvetica 75 typeface. Non-printing keys are shown enclosed in square brackets:

[P] As a command, with no corresponding displayed character
[ESC]

General narrative and descriptive text will be shown in one of a variety of serif typestyles, eg: Century, Palatino, Times etc.

HiSoft DEVPAC for the CPC464

Introduction

Welcome to the land of low-level assembler programming! A land where there are many rewards for the patient programmer (fast, compact programs) and many, many pits into which the unwary programmer may (will) fall.

In this section we shall attempt to give you a taste of the benefits that can be reaped from careful assembly language programming. If you are already experienced in this type of programming, then please feel free to skip this section and move on to the other Sections which give the technical details of Hisoft Devpac.

Hisoft Devpac is a suite of two programs; *GENA3* - a Z80 assembler and *MONA3* - a disassembler/debugger. Firstly, what on earth does that mean?

As you undoubtedly know, computers think in binary - all the computer's memory and all the instructions that it obeys are just a string of 1's and Ø's or on's and off's.

A high-level language, like BASIC or Pascal, groups these machine instructions together to form English-like, commands such as `PRINT`. The advantage of this is that these high-level commands are easy to understand and therefore relatively easy to use. The disadvantage is that one high-level command may be equivalent to hundreds, or even thousands, of machine, or low-level, instructions. Thus programs written in high-level languages can be slow to run and large in size, although they are generally very easy to write. So, if we want to be able to write fast and compact programs, it would seem advantageous to be able to produce low-level programs consisting of machine-code instructions - instructions that the computer's brain understands directly. But, as we said above, these instructions are just a string of 1's and Ø's (or bits, BInary digiTs) and nobody wants to have to remember (or even look up) the pattern of bits for each instruction. You don't have to: assembly language comes to the rescue by providing reasonably English-like words for each machine-code instruction. So you can write a low-level language program by using mnemonics that correspond one-for-one to machine-code instructions. Then you must convert your assembly-language program into machine-code; you do this with an assembler. *GENA3* is an assembler for the Z80 microprocessor and it contains its own editor that enables you to create the assembler program and then assemble it (convert it to machine-code) in a very simple and interactive manner.

Occasionally, you might wish to look at a piece of machine-code and try to understand how it works. To do this you may want to convert the machine-code into the corresponding assembler mnemonics so that it is easier to understand - a program that does this backwards conversion is called a disassembler. You may want other facilities to help you to understand the low-level program such as being able to see how each instruction affects the state of the processor in the computer -so you may want to single-step those instructions one at a time, seeing how the Z80's registers and flags (the working store of the processor) change as each instruction is executed. You may also want to run a number of instructions and then stop and look at the machine state i.e. set a breakpoint in the program so that execution breaks at that point. A package that lets you do all these things is called a debugger - it helps you to get rid of all those horrible little bugs. *MONA3* is a combined disassembler/debugger which gives you many powerful commands to help you explore and understand machine-code programs.

So that is what Devpac gives you - confused? We hope not. Impatient? Ok let's produce an assembler program:

Load your Devpac cassette into the CPC464 tape deck, press the button marked **[PLAY]** and then type:

`RUN "` **[ENTER]**

on the keyboard.

After a short while, the message

```
Load Address?
```

will appear on the screen. Leave the tape deck as it is and type:

`1000` **[ENTER]**

on the keyboard. The assembler will now be loaded into the memory of the CPC464, starting at address `1000`. When the loading is finished (it should take roughly 135 seconds) the assembler will run automatically and produce a sign-on message on the screen and a '>' sign followed by the normal block cursor. Press **[CAPS LOCK]** and then type:

`I10,10` **[ENTER]**

and you will be greeted with the number `10` at the left side of the screen followed by a space. We can now enter some assembler mnemonics finishing each line with **[ENTER]** and the lines will be entered into the program at the line numbers given on the left. So, following the `10`, type:

```
ENT $ [ENTER]
LD B,26  [ENTER]
LD A,"A" [ENTER]
LOOP: CALL #BB5A [ENTER]
INC A [ENTER]
DJNZ LOOP [ENTER]
RET [ENTER]
```

and finally type **[CTRL]C** - this means hold the **[CTRL]** key down and press **C**. You will now be back in the editor's command mode with the '>' prompt again. List your program by typing:

```
L [ENTER]
```

Notice how the listing is neatly tabulated.

Well, what does this program do and how to we make it work?

Let's look at the program line by line:

```
10 ENT $
```

This simply tells the editor that, if you want to run the program, then the program should be entered at this point. ENT does not generate any machine code - it is what is called an assembler directive.

```
20 LD B,26
```

This puts the value 26 in one of the Z80 registers - register B.

```
30 LD A,"A"
```

This puts the value of the ASCII character 'A' in the Z80's A register.

```
40 LOOP: CALL #BB5A
```

A CALL instruction is used to invoke a subroutine somewhere else in the machine. The subroutine at #BB5A calls a routine in the operating system which prints out on the screen whatever character is in the A register and then returns to the instruction following the CALL. So this should print out 'A'.

```
50 INC A
```

This simply increments by one the value held in the A register. It so happens that ASCII characters have sequential values, so that 'B' comes after 'A'. Thus register A is increased to hold the value 'B'.

```
60 DJNZ LOOP
```

A fairly powerful instruction this. It says: decrement the B register by one and, if not zero, then goto LOOP. So we can use this instruction to do something many times - a bit like a FOR..NEXT loop. Here register B was 26 so DJNZ LOOP decrements it to 25 and then jumps to LOOP (since B is non-zero). Now the instruction at LOOP prints out the value in the A register so that 'B' is now printed on the screen. Then register A is incremented again (to 'C') and we come to DJNZ LOOP again. So we will keep going round the loop 26 times until register B is equal to zero and then we will drop through to:

```
7Ø RET
```

This RETurns to the editor. Generally RET returns from a subroutine. In this case we were not in a subroutine but if we run the program from the editor then it is treated as a subroutine and so control returns to the editor.

Right, what about running it then? Well, at the moment, all that we have is the source text of our program - we haven't actually converted it to machine-code yet. We do that with the editor command **'A'** (for Assemble). So type:

A [ENTER]

You will see the message:

```
Table size:
```

appear. In most cases you do not have to enter a value here so just press **[ENTER]**. Now the message:

```
Options:
```

appears. There are many assembly options but the default will do us here - so just press **[ENTER]** again. You will now get a listing on the screen, the assembly listing. The main body of this tells you the machine-code corresponding to your assembler program and where that machine code has been placed in memory. The first 4 characters tell you the memory address (in hexadecimal, base 16) of your machine-code and these are followed by the actual machine-code. So you can see that LD B,26 generates Ø61A in the machine. This Ø61A is actually another representation of the actual bits set, each character corresponds to 4 bits e.g. Ø is ØØØØ, 6 is Ø11Ø, 1 is ØØØ1 and A is 1Ø1Ø. But you don't need to worry about all that; the important thing is that you have now produced your machine-code in memory. You can run it very simply, just type:

R [ENTER]

What happened? You should have seen the alphabet printed. Run it again (press **R [ENTER]**) .. and again. Easy?

Now change it. Type:

2Ø LD B,6Ø [ENTER]
A [ENTER]

Press **[ENTER]** after Table size? and Options? and then run the new program by typing:

R [ENTER]

This time you got 60 characters displayed.

Assemblers are intended to make the task of the low-level programmer easier and *GENA3* contains many features that enable you to create more easily understood programs. Let's delete the above program and re-type it in a more readable form. Delete it by typing:

```
D1Ø,7Ø [ENTER]
```

Note that this has only deleted the source of the program (the assembler mnemonics etc.); it has not deleted the machine-code from memory (try the **R** command again). Now enter a new program:

```
I1Ø,1Ø [ENTER]
LENGTH: EQU 26 [ENTER]
FIRST: EQU "A" [ENTER]
CONOUT: EQU #BB5A [ENTER]
[ENTER]
ENT $ [ENTER]
LD B,LENGTH [ENTER]
LD A,FIRST [ENTER]
Loop: CALL CONOUT [ENTER]
INC A [ENTER]
DJNZ Loop [ENTER]
RET [ENTER]
[CTRL/C]
```

List the program (using **L [ENTER]**) - it is more readable isn't it? We have used the `EQU` assembler directive (remember directives don't generate any machine code) to assign values to variable-like names. They are not really variables because you cannot change them at any time but they do allow the program to have more meaning. Now assemble the program (**A [ENTER]**), pressing **[ENTER]** after Table size: and Options: and run it (**R [ENTER]**). It produces the same results as the earlier, more primitive program.

Well that's about as much as we can do without writing a whole book on the subject and that's been done before! If you are new to assembly language then you should buy (or borrow) a good book and use it in conjunction with the rest of this manual. The classic book is 'Programming the Z80' by Rodney Zaks although this can be heavy going for absolute beginners. With the advent of inexpensive micros, books on assembly language are many and varied - just remember that you need one on Z80 assembly language and not `65Ø2`, `68ØØØ` etc. etc. Good luck and, just to show you why people use machine-code, delete your program by:

```
D1Ø,11Ø [ENTER]
```

and enter this one:

```
I10,10 [ENTER]
ENT $ [ENTER]
COUNT: EQU 10000 [ENTER]
PLOT: EQU #BBEA [ENTER]
MODE: EQU #BC0E [ENTER]
GRPEN: EQU #BBDE [ENTER]
[ENTER]
LD A,1 [ENTER]
CALL MODE [ENTER]
LD HL,300 [ENTER]
LD (RANUM),HL [ENTER]
LD BC,COUNT [ENTER]
[ENTER]
LOOP: PUSH BC [ENTER]
BIT 4,C [ENTER]
JP Z,NOSET [ENTER]
LD A,C [ENTER]
CALL GRPEN [ENTER]
NOSET:CALL RANDOM [ENTER]
PUSH HL [ENTER]
CALL RANDOM [ENTER]
POP DE [ENTER]
CALL PLOT [ENTER]
POP BC [ENTER]
DEC BC [ENTER]
LD A,B [ENTER]
OR C [ENTER]
JP NZ,LOOP [ENTER]
RET [ENTER]
[ENTER]
RANDOM: LD DE,(RANUM) [ENTER]
LD H,E [ENTER]
LD L,-3 [ENTER]
LD A,D [ENTER]
SBC HL,DE [ENTER]
SBC A,0 [ENTER]
SBC HL,DE [ENTER]
SBC A,0 [ENTER]
LD E,A [ENTER]
LD D,0 [ENTER]
SBC HL,DE [ENTER]
JP NC,RAN1 [ENTER]
INC HL [ENTER]
RAN1: LD (RANUM),HL [ENTER]
LD A,H [ENTER]
AND %00000001 [ENTER]
LD H,A [ENTER]
RET [ENTER]
[ENTER]
RANUM: DEFS 2 [ENTER]
[CTRL/C]
```

Now assemble this program (**A [ENTER]**) - you might like to use option 4 (enter **4 [ENTER]** after the `Options:` message) to turn off the assembly listing and run it using **R [ENTER]**. You should find that the plotting speed is something like `1500` points per second. We could make it even faster by accessing the screen memory directly. But let's look at a similar program in BASIC. From within the assembler type:

`B` **[ENTER]**

This will take you back into BASIC, so type:

```
NEW [ENTER]
AUTO [ENTER]
CLS [ENTER]
J%=1 [ENTER]
FOR I%=1 TO 10000 [ENTER]
X%=RND*639 : Y%=RND*399 [ENTER]
IF (I%MOD 20)=0 THEN J%=INT(RND*3)+1 [ENTER].
PLOT X%,Y%,J% [ENTER]
NEXT I% [ENTER]
[ESC]
RUN [ENTER]
```

The result is far slower than the machine-code routine, by roughly 20 times. Nevertheless, this is actually a good speed for BASIC and you can normally expect machine-code programs to run up to 1000 times faster than the equivalent BASIC program. However, as you can see, the machine-code version is much more difficult to understand - you always have to compromise!

For information, here are the times of this 'Night Sky' routine written in BASIC, Pascal and machine-code:

150 seconds, 22 seconds and 8 seconds respectively.

We hope that this section has whetted your appetite for assembler-language programming and we urge you to work through the rest of this manual slowly and carefully, trying things out as you go along. There is an extended example of using the assembler in Appendix 3 and an example of using the disassembler/debugger in Section 2.11 of the *MONA3* manual.

Assembler and Editor - SECTION 1

Note: Throughout the Assembler Editor manual **[ESC]** is equivalent to **[CTRL]C** ie. you may use either **[ESC]** or **[CTRL]C** to achieve the same result.

GENA3 is a powerful and easy-to-use Z80 assembler which is very close to the standard Zilog assembler in definition. Unlike many other assemblers available for microcomputers, *GENA3* is an extensive, professional piece of software and you are urged to study the following sections, together with the example in Appendix 3, very carefully before attempting to use the assembler. If you are a complete novice, work through Appendix 3 first, and/or Section 0 first.

GENA3 is roughly 7K bytes in length, once relocated. It contains its own integral line editor which places the textfile immediately after the *GENA3* code while the assembler's symbol table is created after the textfile. Thus when loading *GENA3* you must allow enough room to include the assembler itself and the maximum symbol table and text size that you are likely to use in the current session. It will often be convenient, therefore, to load *GENA3* into low memory.

To load *GENA3* into your Amstrad computer, simply type:

`RUN"`**[ENTER]**

and press **[PLAY]** on the tape deck. Firstly, a short BASIC program will be loaded and this will automatically run itself and produce the message:

```
Load Address?
```

You should enter a decimal number between `1000` and `30000` and then press **[ENTER]**. The number you enter is the address at which the assembler will be loaded - a good address is `1000`. After you have entered this number the message:

```
Please wait..loading GENA3.1..
```

will appear on the screen and the tape recorder will be started again automatically. The assembler is now being loaded into the computer; this will take roughly 130 seconds. When *GENA3* has been fully loaded loaded, it will run itself and produce a sign-on message, and a 'Help' screen showing the commands invoked by capital letters.

Note: if you intend to have both *GENA3*, the assembler, and *MONA3*, the disassembler/debugger, in the computer at the same time, then always load *GENA3* in low memory (say at `1000`) and *MONA3* in high memory (say at `30000`). In this case *GENA3* should be loaded last.

SECTION 2
Details of GENA3.

2.0 How *GENA3* Works.

GENA3 is a fast, two-pass Z80 assembler which is easily adaptable to run on most Z80 systems. The assembler assembles all standard Z80 mnemonics and has added features which include conditional assembly, many assembler commands and a binary-tree symbol table.

When you invoke an assembly (using the editor '**A**' command - see section 3) you will first be asked to specify '`Table size:`' in decimal. This is the amount of space that will be allocated to the symbol table during the assembly. If you default (by simply hitting **[ENTER]**) then *GENA3* will choose a symbol table size that it thinks is suitable for the size of the text - normally this will be perfectly acceptable. Note that when using the 'include' option you may have to specify a larger than normal symbol table size; the assembler cannot predict the size of the file that will be included.

After '`Table size:`' you will be asked for any '`Options:`' that you require. Enter these in decimal adding the option numbers together if you want more than one option. The options available are:

Option `1`	Produce a symbol table listing at the end of the second pass of the assembly
Option `2`	Do not generate any object code.
Option `4`	Do not produce an assembly listing.
Option `8`	Direct any assembly listing to the printer.
Option `16`	Simply place the object code, if generated, after the symbol table. The Location Counter is still updated by the ORG so that object code can be placed in one section of memory but designed to run elsewhere.
Option `32`	Turn off the check of where the object code is going - useful for speeding up assembly.

Example: Option `36` produces a fast assembly - no listing is generated and no checks are made to see where the object code is being placed.

Note that if you have used Option `16` then the ENT assembler directive will have no effect. You can work out where the object code has been placed if Option `16` has been specified by using the editor '**X**' command to find out the end of the text (the second number displayed) and then adding to this the amount of symbol table allocated + 2.

Assembly takes place in two passes; during the first pass *GENA3* searches for errors and compiles the symbol table, the second pass generates object code (if option `2` is not specified). During the first pass nothing is displayed on the screen or printer unless an error is detected, in which case the rogue line will be displayed with an error number below it (see Appendix 1). The assembly is paused - press **[CTRL]C** to return to the editor or any other key to continue the assembly from the next line.

At the end of the first pass the message:

```
Pass 1 errors: nn
```

will be displayed. If any errors have been detected the assembly will then halt and not proceed to the second pass. If any labels were referenced in the operand field but never declared in a label field then the message '`*WARNING* label absent`' will be displayed for each missing label declaration.

It is during the second pass that object code is generated (unless generation has been turned off by Option `2` - see above). An assembler listing is generated during this pass unless it has been switched off by Option `4` or the assembler command **`*L-`**. The assembler listing is normally of the form:

```
CØØØ  21Ø1ØØ   25   label  LD   HL,1
1     6        15   21     28   33
```

The first entry in the line is the value of the Location Counter at the start of processing this line, unless the mnemonic in this line is one of the pseudo-mnemonics `ORG`, `EQU` or `ENT` (see Section 2.6) in which case the first entry will represent the value in the operand field of the instruction. This entry is normally displayed in hexadecimal but may be displayed in unsigned decimal through use of the assembler command **`*D+`** (see Section 2.8).

The next entry, from column 6, is up to 8 characters in length (representing up to 4 bytes) and is the object code produced by the current instruction.

Then comes the line number - an integer in the range 1 to 32767 inclusive.

Columns 21 – 26 of the line contain the first 6 characters of any label defined in this line, unless the label is present alone in the line in which case the whole label will be displayed.

Then follows the mnemonic, operands (if any) and comments.

The assembly listing may be paused at the end of a line by hitting any key - subsequently hit **[CTRL]C** to return to the editor or any other key to continue the listing.

The only errors that can occur during the second pass are `*ERROR* 1Ø` (see Appendix 1) and '`Bad ORG!`' (which occurs when the object code will overwrite *GENA3*, the textfile or the symbol table - the detection of this can be turned off by Option `32`). `*ERROR* 1Ø` is non-fatal and you may continue the assembly as for first pass errors whereas '`Bad ORG!`' is fatal and immediately returns control to the editor. A '`**File Closed**`' warning may occur if an ORG directive is used while saving the object code to tape via the **`*T`** assembler command - this warns that the current tape object file has been closed and the **`*T`** command turned off.

At the end of the second pass the message:

```
Pass 2 errors:nn
```

will be displayed followed by warnings of any absent labels - see above. The following message is now displayed:

```
Table used: xxxxx from yyyyy
```

This informs you of how much of the symbol table was used compared with how much was allocated.

At this point, if the assembler directive `ENT` has been used correctly, the message '`Executes:` nnnnn'
is displayed. This shows the run address of the object code -you can execute the code by using the editor '**R**' command. Be careful using the '**R**' command unless you have just finished a successful assembly and seen the '`Executes:` nnnnn' message.

Finally, if option `1` has been specified, an alphabetic list of the labels used and their associated values will be produced.

Control now returns to the editor.

2.1 Assembler Statement Format.

Each line of text that is to be processed by *GENA3* should have the following format where certain fields are optional:

```
LABEL:   MNEMONIC      OPERANDS        COMMENT
start:   LD            HL,label    ;load HL with label
```

Spaces and tab characters (inserted by the editor) are generally ignored.

The line is processed in the following way:

The first character of the line is inspected and the following action is taken if this character is a '`;`', '`*`' or end-of-line character:

'`;`'	the whole line is treated as a comment i.e. effectively ignored.
'`*`'	expects the next character(s) to constitute an assembler command (see Section 2.8). Treats all characters after the command as a comment.
(end-of-line character)	simply ignores the line.

If the first character is any character other than those listed above then the line is searched for a space, tab character or colon '`:`'. If a colon is found first then all the preceeding characters are taken as making up a label, otherwise it is assumed that the first non-space, non-tab character is the first character of the mnemonic.

Note that, if a mnemonic follows a label, then only the first 6 characters of the label are displayed on the listing. However, if a label is present alone on a line, then all characters of the label are displayed although only the first 6 will be entered into the symbol table.

After processing a valid label, or if no label is detected, the assembler searches for the next non-space/tab character and expects this to be either an end-of-line character or the start of a Z80 mnemonic (see Appendix 2) of up to 4 characters in length and terminated by a space/tab or end-of-line character. If the mnemonic is valid and requires one or more operands then spaces/tabs are skipped and the operand field is processed.

Labels may be present alone in an assembler statement; useful for increasing the readability of the listing. Labels which are present alone in a line will not be truncated to 6 characters although only the first 6 characters will be entered into the symbol table.

Comments may occur anywhere after the operand field or, if a mnemonic takes no arguments, after the mnemonic field. Comments are automatically tabbed to column 40 on both the editor and assembler listings.

The mnemonic must always be followed by a space, **[TAB]** or **[ENTER]** character.

2.2 Labels.

A label is a symbol which represents up to 16 bits of information.

A label can be used to specify the address of a particular instruction or data area or it can be used as a constant via the `EQU` directive (see Section 2.6).

If a label is associated with a value greater than 8 bits and it is then used in a context where an 8 bit constant is applicable then the assembler will generate an error message e.g.

```
label:  EQU    #1234
        LD     A,label
```

will cause `*ERROR* 1Ø` to be generated when the second statement is processed during the second pass.

A label may contain any number of valid characters (see below) although only the first 6 are treated as significant; these first 6 characters must be unique since a label cannot be re-defined (`*ERROR* 4`). A label must be terminated by a colon, '`:`', and must not constitute a Reserved Word (see Appendix 2) although a Reserved Word may be embedded as part of a label.

The characters which may be legally used within a label are `Ø - 9`, `$` and `A - z`. Note that '`A - z`' includes all the upper and lower case alphabetics together with the characters `[`, `\`, `]`, `^` , `'` and `_`. A label must begin with an alphabetic character.

Some examples of valid labels are:

```
LOOP:
loop:
a_long_label:
L[1]:
L[2]:
a:
LDIR:     LDIR is not a Reserved Word.
two^5:
```

2.3 Location Counter.

The assembler maintains a Location Counter so that a symbol in the label field can be associated with an address and entered into the Symbol Table. This Location Counter may be set to any value via the ORG assembler directive (see Section 2.6).

The symbol '`$`' can be used to refer to the current value of the Location Counter e.g. `LD HL,$+5` would generate code that would load the register pair `HL` with a value 5 greater than the current Location Counter value.

2.4 Symbol Table.

When a label is encountered for the first time it is entered into a table along with two pointers which indicate, at a later time, how this label is related alphabetically to other labels within the table. If the first occurrence of the label is in the label field then its value (as given by the Location Counter or the value of the expression after an `EQU` assembler directive) is entered into the Symbol Table. Otherwise the value is entered whenever the symbol is subsequently found in the label field.

This type of symbol table is called a Binary Tree Symbol Table and its structure enables symbols to be entered into and recovered from the table in a very short time - essential for large programs. The size of an entry in the table varies from 8 bytes to 13 bytes depending on the length of the symbol.

If, during the first pass, a symbol is defined more than once then an error (`*ERROR* 4`) will be generated since the assembler does not know which value should be associated with the symbol.

If a symbol is never associated with a value then the message '`*WARNING*` symbol absent' will be generated at the end of the assembly. The absence of a symbol definition does not prevent the assembly from continuing.

Note that only the first 6 characters of a symbol are entered into the Symbol Table in order to keep down the size of the table.

At the end of the assembly you will be given a message stating how much memory was used by the Symbol Table during this assembly - you may change how much memory is allocated to the Symbol Table by responding to the 'Table:' prompt when starting the assembly (see Section 2.0).

2.5 Expressions.

An expression is an operand entry consisting of either a single *TERM* or a combination of *TERM*s each separated by an *OPERATOR*. The definitions of *TERM* and *OPERATOR* follow:

TERM:

decimal constant e.g.	`1Ø29`
hexadecimal constant e.g.	`#4Ø5`
binary constant e.g.	`%1ØØØ1Ø1Ø`

character constant e.g. `"a"`

label e.g. `L1Ø29`

also '$' may be used to denote the current value of the Location Counter.

OPERATOR '+' addition
'–' subtraction
'&' logical `AND`
'@' logical OR
'!' logical `XOR`
'*' integer multiplication
'/' integer division
'?' `MOD` function `( a ? b= a – (a/b)*b)`

Notes: '#' is used to denote the start of a hexadecimal number, '%' for a binary number and '""' for a character constant. When reading a number (decimal, hexadecimal or binary) *GENA3* takes the least significant 16 bits of the number (i.e. MOD 65536) e.g. `7ØØ16` becomes `448Ø` and `#5A2C4` becomes `#A2C4`.

A wide variety of operators are provided but no operator precedence is observed -expressions are evaluated strictly from left to right. The operators '*','/' and ? are provided merely for added convenience and not as part of a full expression handler which would increase the size of *GENA3*.

If an expression is enclosed within parentheses then it is taken as representing a memory address as in the instruction `LD HL`,(loc+5) which would load the register pair `HL` with the 16 bit value contained at memory location 'loc+5'.

Certain Z80 instructions (`JR` and `DJNZ`) expect operands which have an 8 bit value and not a 16 bit one - this is called relative addressing. When relative addresses are specified *GENA3* automatically subtracts the value of the Location Counter at the next instruction from the value given in the operand field of the current instruction in order to obtain the relative address for the current instruction. The range of values allowed as a relative address are the Location Counter value of the next instruction -128 to +127.

If, instead, you wish to specify a relative offset from the Location Counter value of the current instruction then you should use the symbol '$' (a Reserved Word) followed by the required displacement. Since this is now relative to the current instruction's Location Counter value the displacement must be in the range -126 to +129 inclusive.

Examples of valid expressions:

```
#5ØØØ - label
%1ØØ11Ø1 ! %1Ø11                  gives %1ØØØ11Ø
#3456? #1ØØØ                      gives #456
4 + 5 * 3 - 8                     gives 19
$ - label + 8
2345 / 7 - 1                      gives 334
"A"+128
"y"-";"+7
(5*label-#1ØØØ & %1111)
17@%1ØØØ                          gives 25
```

Note that spaces may be inserted between `TERM`s and `OPERATOR`s and vice versa but not within `TERM`s.

If a multiplication operation would result in an absolute value greater than `32767` then `*ERROR* 15` is reported while if a division operation involves a division by zero then `*ERROR* 14` is given - otherwise overflow is ignored. All arithmetic uses the two's complement form where any numbers greater than `32767` are treated as negative e.g. `60000` = `-5536` (`60000-65536`).

2.6 Assembler Directives.

Certain 'pseudo-mnemonics' are recognised by *GENA3*. These assembler directives, as they are called, have no effect on the Z80 processor at run - time i.e. they are not decoded into opcodes, they simply direct the assembler to take certain actions at assembly time. These actions have the effect of changing, in some way, the object code produced by *GENA3*.

Pseudo-mnemonics are assembled exactly like executable instructions; they may be preceded by a label (necessary for `EQU`) and followed by a comment. The directives available are:

`ORG` expression

sets the Location Counter to the value of 'expression'. If option `2` and option `16` are both not selected and an `ORG` would result in the overwriting of the *GENA3* program, the textfile or the symbol table then the message '`Bad ORG!`' is displayed and the assembly is aborted. See Section 2.0 for more details on how options `2` and `16` affect the use of `ORG`.

`EQU` expression

must be preceded by a label. Sets the value of the label to the value of 'expression'. The expression cannot contain a symbol which has not yet been assigned a value (`*ERROR* 13`).

`DEFB` expression,expression,....

each 'expression' must evaluate to 8 bits; the byte at the address currently held by the Location Counter is set to the value of 'expression' and the Location Counter advanced by 1. Repeats for each expression.

`DEFW` expression,expression,....

sets the 'word' (two bytes) at the address currently held by the Location Counter to the value of 'expression' and advances the Location Counter by 2. The less significant byte is placed first followed by the more significant byte. Repeats for each expression.

`DEFS` expression

increases the Location Counter by the value of 'expression' - equivalent to reserving a block of memory of size equal to the value of 'expression'. Zeroes are written throughout the allocated space.

DEFM "s"

defines the contents of n bytes of memory to be equal to the ASCII representation of the string s, where n is the length of the string and may be, in theory, in the range 1 to 255 inclusive although, in practice, the length of the string is limited by the length of the line you can enter from the editor. The first character in the operand field ('"' above) is taken as the string delimiter and the string s is defined as those characters between two delimiters; the end-of-line character also acts as a terminator of the string.

ENT expression

sets the execute address of the assembled object code to the value of expression -used in conjunction with the editor 'R' command (see Section 3). There is no default for the execute address.

2.7 Conditional Pseudo - mnemonics.

Conditional pseudo-mnemonics provide the programmer with the capability of including or not including certain sections of source text in the assembly process. This is made available through the use of IF, ELSE and END.

IF expression

this evaluates 'expression'. If the result is zero then the assembly of subsequent lines is turned off until either an 'ELSE' or an 'END' pseudo-mnemonic is encountered. If the value of 'expression' is non-zero then the assembly continues normally.

ELSE

this pseudo-mnemonic simply flips the assembly on and off. If the assembly is on before the 'ELSE' is encountered then it will subsequently be turned off and vice versa.

END

'END' simply turns the assembly on.

Note: Conditional pseudo-mnemonics cannot be nested; no check is made for nested IFs so any attempt to nest these mnemonics will have unspecified results.

2.8 Assembler Commands.

Assembler commands, like assembler directives, have no effect on the Z80 processor at runtime since they are not decoded into opcodes. However, unlike assembler directives, they also have no effect on the object code produced by the assembler -assembler commands simply modify the listing format.

An assembler command is a line of the source text that begins with an asterisk '*'. The letter after the asterisk determines the type of the command and must be in upper case. the remainder of the line may be any text except that the commands 'L', 'T' and 'D' expect a '+' or a '–' after the command.

The following commands are available:

***E**

(eject) causes three blank lines to be sent to the screen or a new page character to be sent to the printer - useful for separating modules.

***Hs**

causes string s to be taken as a heading which is printed after each eject (***E**). ***H** automatically performs a ***E.**

***S**

causes the listing to be stopped at this line. The listing may be reactivated by pressing any key on the keyboard. Useful for reading addresses in the middle of the listing. Note: ***S** is still recognised after a ***L-**, ***S** does not halt printing.

***L−**

causes listing and printing to be turned off beginning with this line.

***L+**

causes listing and printing to be turned on starting with this line.

***D+**

causes the value of the Location Counter to be given in decimal at the beginning of each line instead of the normal hexadecimal .Unsigned decimal is used

***D−**

reverts to using hexadecimal for the value of the Location Counter at the start of each line.

***F** filename

This is a very powerful command which allows you to assemble text from tape - the textfile is read from the tape into a buffer, a block at a time, and then assembled from the buffer; this allows you to create large amounts of object code since the text being assembled does not take up valuable memory space.

The filename (up to 8 characters) of the textfile you wish to 'include' at this point in the assembly may, optionally, be specified after the '**F**' and must be preceded with a space. If no filename is given then the first textfile found on the tape is included.

Any textfile that you wish to include via this option must have been previously dumped to tape using the editor's '**P**' command.

Whenever the assembler detects an '**F**' command it asks you to '`Press PLAY then any key`' and this will happen in the first and second passes since the include text must be scanned in each pass. The tape is then searched for an include file with the required filename, or for the first file. If an include file is found whose filename does not match that required then the message '`Found filename`' is displayed and searching continues, otherwise '`Loading filename`' is displayed, the file loaded, block by block, and included.

See Appendix 3 for an example of the use of this command.

`*T+`   filename

This command is used to dump object code to tape (under the given filename) while the assembly is taking place. This output of object code may be halted by a *T-command, an `ORG` directive or the end of the assembly. The code so saved may be re-loaded using the debugger *MONA3*.

`*T-`

Closes the current object code tape file.

Assembler commands, other than *F, are recognised only within the second pass.

If assembly has been turned off by one of the conditional pseudo-mnemonics then the effect of any assembler command is also turned off.

SECTION 3

The Integral Editor.

3.1 Introduction to the Editor.

The editor supplied with all versions of *GENA3* is a simple, line - based editor designed to work with all **Z80** operating systems while maintaining ease of use and the ability to edit programs quickly and efficiently.

In order to reduce the size of the textfile, a certain amount of compression of spaces is performed by the editor. This takes place according to the following scheme: whenever a line is typed in from the keyboard it is entered, character by character into a buffer internal to the assembler; then, when the line is finished (i.e. you hit **[-ENTER]**), it is transferred from the buffer into the textfile. It is during this transfer that certain spaces are compressed: the line is scanned from its first character, if this is a space then a tab character is entered into the textfile and all subsequent spaces are skipped. If the first character is not a space then characters are transferred from the buffer to the textfile until a space is detected whereupon the action taken is the same as if the next character was the first character in the line. This is then repeated a further time with the result that tab characters are inserted at the front of the line or between the label and the mnemonic and between the mnemonic and the operands. Of course, if any carriage return **[ENTER]** character is detected at any time then the transfer is finished and control returned to the editor.

This compression process is transparent to the user who may simply use the **[TAB]** key to produce a neatly tabulated textfile which, at the same time, is economic on storage.

Note that spaces are not compressed within comments and spaces should not be present within a label, mnemonic or operand field.

The Editor is entered automatically when *GENA3* is executed and displays the message:

```
Hisoft GENA3.1 Assembler
Copyright Hisoft 1984
All rights reserved
```

followed by the editor prompt '>'.

In response to the prompt you may enter a command line of the following format:

`C N1,N2,S1,S2` followed by **[ENTER]**

`C` is the command to be executed (see Section 3.2 below).

`N1` is a number in the range 1 - 32767 inclusive.

`N2` is a number in the range 1 - 32767 inclusive.

`S1` is a string of characters with a maximum length of 20.

`S2` is a string of characters with a maximum length of 20.

The comma is used to separate the various arguments (although this can be changed - see the '**S**' command) and spaces are ignored, except within the strings. None of the arguments are mandatory although some of the commands (e.g. the '**D**'elete command) will not proceed without `N1` and `N2` being specified. The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of `N1` and `N2` are initially set to 10 and the strings are initially empty. If you enter an illegal command line such as `F-1,100,HELLO` then the line will be ignored and the message '`Pardon?`' displayed - you should then retype the line correctly e.g. `F1,100,HELLO`. This error message will also be displayed if the length of S2 exceeds 20; if the length of S1 is greater than 20 then any excess characters are ignored.

Commands may be entered in upper or lower case.

While entering a command line, certain control functions may be used e.g. **[CTRL]X** to delete to the beginning of the line, **[TAB]** to advance the cursor to the next tab position.

The following sub-section details the various commands available within the editor - note that wherever an argument is enclosed by the symbols '< >' then that argument must be present for the command to proceed.

The Editor Commands.

3.2.1 Text Insertion.

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the 'I' command. Note that if you type a line number followed by **[ENTER]** (i.e. without any text) then that line will be deleted from the text if it exists. Whenever text is being entered then the control functions **[CTRL]X** (delete to the beginning of the line), **[TAB]** (go to the next tab position) and **[CTRL]C** (return to the command loop) may be employed.

The **[DEL]** key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within *GENA3* and if this buffer should become full then you will be prevented from entering any more text -you must then use **[DEL]** or **[CTRL]X** to free space in the buffer.

If, during text insertion, the editor detects that the end of text is nearing the top of RAM it displays the message '`Bad Memory!`'. This indicates that no more text can be inserted and that the current textfile, or at least part of it, should be saved to tape for later retrieval.

Command: **I** n,m

Use of this command gains entry to the automatic insert mode: you are prompted with line numbers starting at n and incrementing in steps of m. You enter the required text after the displayed line number, using the various control codes if desired and terminating the text line with **[ENTER]**. To exit from this mode use **[CTRL]C**.

If you enter a line with a line number that already exists in the text then the existing line will be renumbered one greater than it was and the new line inserted before it, **[ENTER]**. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

If, when typing in text, you get to the end of a screen line without having entered 80 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line.

3.2.2 Text Listing.

Command: **L** n,m

This lists the current text to the display device from line number n to line number m inclusive. The default value for n is always 1 and the default value for m is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire textfile simply use '**L**' without any arguments. Tabulation of the line is automatic, resulting in a clear separation of the various fields with the line. The number of screen lines listed on the display device is always 24. After listing 24 lines the list will pause (if not yet at line number m), hit **[CTRL]C** to return to the main editor loop or any other key to continue the listing.

3.2.3 Text Editing.

Once some text has been created there will inevitably be a need to edit some lines. Many commands are provided to enable lines to be amended, deleted, moved and renumbered. Various commands exist to achieve this and these are given below. Some elementary form of screen editing is also supported and this works as follows:

Whenever you are in the command mode of the editor (i.e. with a '**>**' sign in the left margin of the current line) then you may split the cursor into a read cursor and a write cursor by holding the **[SHIFT]** key down together with one of the cursor keys. The write cursor will remain in the same position as the original cursor while you can move the read cursor about the screen (but not off the screen) using **[SHIFT]** and the cursor keys. Release **[SHIFT]** and the relevant cursor key when the read cursor is where you want it to be. You can now either type directly onto the keyboard and the characters will appear at the write cursor or you can press the **[COPY]** key and in this case characters will be transferred from the read cursor position to the write cursor position and both cursor positions will be incremented.

To terminate this screen copy mode simply press [ENTER], the read cursor will disappear and the line containing the write cursor will be scanned normally by the editor.

In addition to this screen-editing capability, various line-editing commands are supported viz:

Command: **D** ‹n,m›

All lines from n to m inclusive are deleted from the textfile. If m‹n, or less than two arguments are specified, then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n ; this can also be accomplished by simply typing the line number followed by **[ENTER]**.

Command: **M** n,m,d

Moves the block of text between line numbers n and m inclusive to a position before the line with line number d and deletes the original block of text. The block that is moved will be renumbered starting with a line number that is 1 greater than the line number preceeding d.

Lines cannot be moved within themselves so that d must not lie within the block of lines n to m.

Command: **N** <n,m>

Use of the '**N**' command causes the textfile to be renumbered with a first line number of n and in line number steps of m. Both n and m must be present and if the renumbering would cause any line number to exceed 32767 then the original numbering is retained.

Command: **F** n,m,f,s

The text existing within the line range n < x < m is searched for an occurrence of the string f - the 'find' string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see below. You may then use commands within the Edit mode to search for subsequent occurrences of the string f within the defined line range or to substitute the string s (the 'substitute' string) for the current occurrence of f and then search for the next occurrence of f; see below for more details.

Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter '**F**' to initiate the search - see the example in Section 3.3 for clarification.

Command: **E** n

Edit the line with line number n. If n does not exist then no action is taken; otherwise the line is copied into a buffer and displayed on the screen (with the line number), the line number is displayed again underneath the line and the Edit mode is entered. All subsequent editing takes place within the buffer and not in the text itself; thus the original line can be recovered at any time.

In this mode a pointer is imagined moving through the line (starting at the first character) and various sub-commands are supported which allow you to edit the line. The sub-commands are:

'_' (space) - increment the text pointer by one i.e. point to the next character in the line. You cannot step beyond the end of the line.

[DEL] - decrement the text pointer by one to point at the previous character in the line. You cannot step backwards beyond the first character in the line.

[ENTER] - end the edit of this line keeping all the changes made.

Q - quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated.

R - reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally.

L - list the rest of the line being edited i.e. the remainder of the line beyond the current pointer position. You remain in the Edit mode with the pointer re-positioned at the start of the line.

K - kill (delete) the character at the current pointer position.

Z - delete all the characters from (and including) the current pointer position to the end of the line.

F - find the next occurrence of the 'find' string previously defined within a command line (see the '**F**' command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the 'find' string in the current line. If an occurrence of the 'find' string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the found string.

S - substitute the previously defined 'substitute' string for the currently found occurrence of the 'find' string and then perform the sub-command '**F**' i.e. search for the next occurrence of the 'find' string. This, together with the above '**F**' sub-command, is used to step through the textfile optionally replacing occurrences of the 'find' string with the 'substitute' string - see Section 3.3 for an example.

I - insert characters at the current pointer position. You will remain in this sub-mode until you press **[ENTER]** - this will return you to the main Edit mode with the pointer positioned after the last character that you inserted. Using **[DEL]** within this sub -mode will cause the character to the left of the pointer to be deleted from the buffer, A '*' cursor is displayed while in Insert mode.

X - this advances the pointer to the end of the line and automatically enters the insert sub-mode detailed above.

C - change sub - mode. This allows you to overwrite the character at the current pointer position and then advances the pointer by one. You remain in the change sub -mode until you press **[ENTER]** whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. **[DEL]** within this sub-mode simply decrements the pointer by one i.e. moves it left while **[TAB]** has no effect. A '+' cursor is displayed while in change mode.

3.2.4 Tape commands.

Text may be saved to tape or loaded from tape using the commands '**P**', '**V**' and '**G**', while object code may be saved using the '**O**' command.

Command: **Pn,m,s**

The line range defined by n < x < m is saved to tape under the filename specified by the string s. Remember that these arguments may have been set by a previous command. Before entering this command make sure that your datacorder is in `RECORD` mode.

Command: **Q,,s**

This has the same function as the **P** command except that the text is dumped out in ASCII i.e. with line numbers removed and CRLF (carriage return/line feed) at the end of each line.

Command: **G,,s**

The tape is searched for a file with a filename of s; when found, it is loaded at the end of the current text. If a null string is specified as the filename then the first textfile on the tape is loaded. Both ASCII (saved using Q) and non-ASCII (saved using P) files may be loaded using this command since the type of file is automatically detected and allowed for.

After you have entered the '**G**' command, you should press **PLAY** on your recorder. A search is now made for a text file with the specified filename, or the first textfile if a null filename is given. If a match is made then the message:

'`Loading filename`'

is displayed, otherwise '`Found filename`' is shown and the search of the tape continues.

Note that if any textfile is already present in the memory then the textfile that is loaded from tape will be appended to the existing file and the whole file will be renumbered starting with line 1 in steps of 1.

Command: **V,,s**

Verifies a file on tape with the existing textfile in the memory. Displays '`Verified`' or '`Failed!`' depending on the outcome of the verification.

Please note that ASCII files (saved using Q) may not be verified using this command. If you desire to check an ASCII file then you should do so from within BASIC.

Command: **O,,s**

Outputs the object code produced by the latest assembly to tape under the filename s. Use *MONA3* or BASIC to re-load this object code.

Command: **Tn**

Changes the speed of any subsequent cassette dump. '**T**' followed by **[ENTER]** selects the slow cassette speed, whereas '**T**' followed by a number greater than 0, and then **[ENTER]**, selects the fast cassette speed..

3.2.5 Assembling and Running from the Editor.

Command: **A**

This causes the text to be assembled from the first line in the textfile. See Section 2 for further details.

Command: **R**

If the source has been assembled without errors and an execute address has been specified by the use of the `ENT` assembler directive then the **'R'** command may be used to execute the object program. The object program can use a `RET (#C9)` instruction to return to the editor so long as the stack is in the same position at the end of the execution of the program as it was at the beginning. Note that `ENT` will have no effect if Option 16 has been specified for the assembly.

3.2.6 Other Commands.

Command: **H**

Provides a help screen of the various commands (given by CAPITAL letters) available in GENA3.

Command: **B**

This simply returns control to the operating system. To re-enter the assembler use either a cold start (The original load address +2) or a warm start (The original load address +4).

Command: **S**,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma **','** is taken as the delimiter; this may be changed by the use of the **'S'** command to the first character of the specified string **d**. Remember that once you have defined a new delimiter it must be used (even within the **'S'** command) until another one is specified.

Note that the separator may not be a space.

Command: **C**

The **'C'** command displays the current values of the delimiter and N1, N2, S1 and S2 i.e. the two default line numbers and the default strings. This is useful before entering any command in which you are going to use default values, to check that these values are correct.

Command: **Z** n,m

The **'Z'** command causes the section of text between lines *n* and *m* inclusive to be output to the printer. If both n and m are defaulted then the whole textfile will be printed. If no printer is present, then `"No Printer!"` is displayed, and no action is taken. During a list to the printer you may pause the printing at the end of a line by hitting any key. Then press **[CTRL]C** to abort the list or any other key to continue.

Command **Y**n

Sets the number of lines per page on printed output to the value n. This allows you to adjust your printed output for different lengths of printer paper.

Command: **X**

'X' simply causes the start and end address of the textfile to be displayed in decimal. This is useful if you wish to save the text from within BASIC, or if you want to see how much memory you have left after the textfile.

'**X**' is also used in conjunction with MONA3 when using MONA3's disassembler to disassemble to a textfile for use by *GENA3*. To interface a textfile generated by MONA3 to the assembler we must first move that textfile to *GENA3*'s `TEXTSTART`, or generate the file directly at that address.

Now, the first address displayed, when '**X**' is used, is the address of the start of the *GENA3* textfile, which may be empty - this is the address at which the textfile generated by MONA3 must start. So arrange to disassemble the code to start at this address and then note down the 'Text end' address as given by the disassembler; we must now amend *GENA3*'s `TEXTEND` to this address. `TEXTEND` is stored at 'Start of *GENA3*' + 7 so, if we loaded *GENA3* at 1000 then `TEXTEND` would be stored in 1007 and 1008 (low byte first). So, take the value of 'Text end' specified by MONA3 (in hex) and poke the low byte in 1007 and the high byte in 1008 and then enter *GENA3* via a warm start (via 1004 in this case). The disassembled textfile can now be edited and assembled by *GENA3*.

Command: **U**

Simply displays the line number of the last line within the textfile; this is useful for quickly finding out where your textfile ends so that you can append to it.

Command: **W**

'**W**' simply flips the character display between 40 and 80 character modes.

Command: **[CTRL]J**

Enters the debugger MONA3 if this present and has been used at least once. If MONA3 is not present, or has not been used then '**[CTRL]J**' has no effect.

3.3 An Example of the use of the Editor.

Let us assume that you have typed in the following program (using `I10,10`):

```
10  *h 16 BIT RANDOM NUMBERS
20
30  ;INPUT:HLcontains previous randomnumberor seed.
40  ;OUTPUT: HL contains new randon number.
50
60  Random:  PUSH  AF ;save registers
70           PUSH  BC
80           PUSH  HL
90           ADD   HL,HL ;*2
100          ADD   HL,HL ;*4
110          ADD   HL,HL ;*8
120          ADD   HL,HL ;*16
130          ADD   HL,HL ;*32
140          ADD   HL,HL ;*64
150          PIP   BC ;old random number
160          ADD   HL,DE
170          LD    DE,41
180          ADD   HL,DE
190          POP   C ;restore registers
200          POP   AF
210          REY
```

This program has a number of errors which are as follows:

Line 10: a lower case 'h' has been used in the assembler command ***H**.

Line 40: '`randon`' instead of '`random`'.

Line 150: '`PIP`' instead of '`POP`'.

Line 160: needs a comment (not an error - merely style).

Line 210: '`REY`' should be '`RET`'.

Also 2 extra lines of `ADD HL,HL` should be added between lines 140 and 150 and all references to the register pair `DE` in lines 160 to 180 should be to register pair `BC`.

To put all this right we can proceed as follows:

`E10` **[ENTER]** then _ (space) **C**(enter change mode) **H [ENTER] [ENTER]**

`F40,40,randon,random` **[ENTER]** then the **'S'** sub - command.

`I142,2` **[ENTER]**

```
142 ADD HL,HL ;*128

144 ADD HL,HL ;*256
```

[CTRL]C

`F150,150,PIP,POP` **[ENTER]** then the **'S'** sub - command.

`E160` **[ENTER]** then `X__;*257+41` **[ENTER] [ENTER]**

`F160,180,DE,BC` **[ENTER]** then repeated use of the sub - command **'S'**.

`E210` **[ENTER]**_ _ _ _ _ _ _ _ _ _ (10 spaces) **C** (change mode) **T [ENTER] [ENTER]**

`N10,10` **[ENTER]** to renumber the text.

You are strongly recommended to work through the above example actually using the editor.

APPENDIX 1

Error numbers and their meanings.

`*ERROR* 1`	Error in the context of this line.
`*ERROR* 2`	Mnemonic not recognised.
`*ERROR* 3`	Statement badly formed.
`*ERROR* 4`	Symbol defined more than once.
`*ERROR* 5`	This line contains an illegal character i.e. a character which is not valid in a particular context.
`*ERROR* 6`	One of the operands in this line is illegal.
`*ERROR* 7`	A symbol in this line is a Reserved Word.
`*ERROR* 8`	Mismatch of registers.
`*ERROR* 9`	Too many registers in this line.
`*ERROR* 10`	An expression that should evaluate to 8 bits evaluates to more than 8 bits.
`*ERROR* 11`	The instructions JP (IX$\pm$n) and JP (IY$\pm$n) are illegal.
`*ERROR* 12`	Error in the formation of an assembler directive.
`*ERROR* 13`	Illegal forward reference i.e. an `EQU`ate has been made to a symbol which has not yet been defined.
`*ERROR* 14`	Division by zero.
`*ERROR* 15`	Overflow in a multiplication operation.
`Bad ORG!`	An `ORG` has been made to an address that would corrupt *GENA*, its textfile or the Symbol Table. Control returns to the editor.
`Out of Table space!`	Occurs during the first pass if insufficient memory has been allowed for the Symbol Table. Control returns immediately to the editor.
`Bad Memory!`	This is displayed if there is no room for any more text to be inserted i.e. the end of text is near the top of RAM. You should save the current textfile, or part of it, to tape.

APPENDIX 2

Reserved words, mnemonics etc.

The following is a list of the Reserved Words within GENA. These symbols may not be used as labels although they may form part of a label. Note that all the Reserved Words may contain upper and/or lower case letters.

A	B	C	D	E	H	L	I	R	$

AF	AF	BC	DE	HL	IX
IY	SP	NC	Z	NZ	M
P	PE	PO			

There now follows a list of the valid Z80 mnemonics, assembler directives and assembler commands. These may be composed of upper and/or lower case letters.

ADC	ADD	AND	BIT	CALL	CCF	CP	CPD	CPDR
CPI	CPIR	CPL	DAA	DEC	DI	DJNZ	EI	EX
EXX	HALT	IM	IN	INC	IND	INDR	INI	INIR
JP	JR	LD	LDD	LDDR	LDI	LDIR	NEG	NOP
OR	OTDR	OTIR	OUT	OUTD	OUTI	POP	PUSH	RES
RET	RETI	RETN	RL	RLA	RLC	RLCA	RLD	RR
RRA	RRC	RRCA	RRD	RST	SBC	SCF	SET	SLA
SRA	SRL	SUB	XOR					

DEFB	DEFM	DEFS	DEFW	ELSE	END	ENT	EQU	IF
ORG								
*D	*E	*H	*L	*S	*C	*F	*T	

APPENDIX 3

A Worked Example.

There follows an example of a typical session using *GENA3* - if you are a newcomer to the world of assembler programs or if you are simply a little unsure how to use the editor/assembler then we urge you to work through this example carefully. Note that each line is terminated by pressing the key marked **[ENTER]** in the usual way.

Session objective:

To write and test a fast integer multiply routine, the text of which is to be saved to tape using the editor's '**P**' command so that it can easily be 'included' in future programs.

Session workplan:

1. Write the multiply routine as a subroutine and save it to tape using the editor's '**P**' command so that it can be easily retrieved and edited during this session, should bugs be present.

2. De-bug the multiply subroutine, editing as necessary.

3. Save the de-bugged routine to tape, using the editor's '**P**' command so that the routine may be 'included' in other programs.

Stage 1 - write the integer multiply routine.

We use the editor's 'I' command to insert the text using **TAB**, if desired, to obtain a tabulated listing. We do not need to use **TAB**, a list of the text will always perform the tabulation for us. Note that the addresses shown in the example assembler listings that follow may not correspond to those produced on your machine; they serve an illustrative purpose only.

```
>I1Ø,1Ø [ENTER]
 1Ø ;A fast integer multiply
 2Ø ;routine. Multiplies HL
 3Ø ;by DE. Return the result
 4Ø ;in HL. C flag set on an
 5Ø ;overflow.
 6Ø
 7Ø ORG #7FØØ
 8Ø
 9Ø Mult: OR A
100 SBC HL,DE;HL>DE?
11Ø ADD HL,DE
12Ø JR NC,Mu1;yes
13Ø EX DE,HL
14Ø Mu1:OR D
15Ø SCF
16Ø RET NZ;OVERflow if DE>255
17Ø OR E;times 0?
18Ø LD E,D
19Ø JR NZ,MU4;no
2ØØ EX DE,HL;Ø
21Ø RET
22Ø
23Ø ;Main routine.
24Ø
25Ø Mu2:EX DE,HL
26Ø ADD HL,DE
27Ø EX DE,HL
28Ø Mu3:ADD HL,HL
29Ø RET C;overflow
3ØØ Mu4:RRA
31Ø JR NC,Mu3
32Ø OR A
33Ø JR NZ,Mu2
34Ø ADD HL,DE
35Ø RET
36Ø [CTRL]C
>P1Ø,35Ø,Mult [ENTER]
```

The above will create the text of the routine and save it to tape. Remember to have your tape recorder in RECORD mode before issuing the **'P'** command.

Stage 2 - de-bug the routine.

First, let's see if the text assembles correctly. We will use option 6 so that no listing is produced and no object code generated.

```
>A
Table size:[ENTER]      (default the symbol table size)
Options: 6 [ENTER]

Hisoft GENA3.1 Assembler. Page 1.
Pass 1 errors: 00
Pass 2 errors: 00
*WARNING* MU4 absent

Table used: 74 from 156
>
```

We see from this assembly that we have made a mistake in line 190 and entered `MU4` instead of `Mu4` which is the label we wish to branch to. So edit line 190:

```
>F190,190,MU4,Mu4 [ENTER]
   190          JR   NZ,              now use the 'S' sub-command
>
```

Now assemble the text again and you should find that it assembles without errors. So now we must write some code to test the routine:

```
>N300,10          renumber so that we can write some more text
>I10,10
 10  ;Some code to test
 20  ;the Mult routine.
 30
 40  LD HL,50
 50  LD DE,20
 60  CALL Mult ;Multiply
 70  LD A,H ;o/p result
 80  CALL Aout
 90  LD A,L
100  CALL Aout
110  RET
120
130 ;Routine to o/p A in hex
140
150 Aout: PUSH AF
160 RRCA
```

```
170  RRCA
180  RRCA
190  RECA
200  CALL  Nibble
210  POP  AF
220  Nibble:AND  %1111
230  ADD  A,#90
240  DAA
250  ADC  A,#40
260  DAA
270  ;Output  a  character  call
280  CALL  #BB5A
290  RET
300  [CTRL]C
>
```

Now assemble the test routine and the Mult routine together.

```
>A
Table size: [ENTER]
Options:  6 [ENTER]
Hisoft  GENA3.1  Assembler.  Page  1.
268D  190  RECA
*ERROR*  02
hit  any  key  to  continue
Pass  1  errors:  01
Table  used:  88  from  201
>
```

We have an error in our routine; RECA should be RRCA in line 190. So:

```
>E190
190  RECA
190  __________(9 spaces)C[enter change mode]R  [ENTER][ENTER]
>
```

Now assemble again, using simply option 4 (no list), and the text should assemble correctly. Assuming it does, we are now in a position to test the working of our Mult routine so we need to tell the editor where it can execute the code from. We do this with the *ENT* directive:

```
>35  ENT  $  [ENTER]
```

Now assemble the text again and the assembly should terminate correctly with the messages:

```
Table used: 88 from 2Ø2

Executes: 9847

>
```

or something similar. Now we can run our code using the editor's **'R'** command. We should expect it to multiply `5Ø` by `2Ø` producing `1ØØØ` which is #3E8 in hexadecimal.

```
>R [ENTER] ØØ32>
```

It doesn't work! Why not? List the lines `38Ø` to `5ØØ (L38Ø,5ØØ)`. You will see that at line `43Ø` the instruction is an `OR D` followed, effectively, by a `RET NZ`. What this is doing is a logical `OR` between the `D` register and the accumulator `A` and returning with an error flag set (the `C` flag) if the result is non-zero. The object of this is to ensure that `DE<256` so that the multiplication does not overflow - it does this by checking that `D` is zero ... but the `OR` will only work correctly in this case if the accumulator `A` is zero to start with, and we have no guarantee that this is so.

We must ensure that `A` is zero before doing the `OR D`, otherwise we will get unpredictable overflow with the higher number returned as the result. From inspection of the code we see that the `OR A` at line `38Ø` could be made into a `XOR A` thus setting the flags for the `SBC HL,DE` instruction and setting `A` to zero. So:

```
>E38Ø [ENTER]

38Ø Mult: OR A

38Ø ________(8 spaces) I        [enter insert mode] X [ENTER] [ENTER]


>
```

Now assemble again (option 4) and run the code, using **'R'**. The answer should now be correct - #3E8.

We can further check the routine by editing lines 40 and 50 to multiply different numbers and then assembling and running - you should find that the routine works perfectly.

Now we have perfected the routine we can save it to tape.

```
>P3ØØ,999,Mult [ENTER]
```

Remember to start the recorder in **[REC]**ord mode before pressing **[ENTER]**. Once the routine has been saved like this it may be included in a program as shown below:

```
500 RET
510
520 ;include the mult routine here
530
540 *F Mult
550
560 ;The next routine
```

When the above text is assembled the assembler will ask you to 'Press PLAY....' when it gets to line 540 on both the first and second pass. Therefore you should have the Mult dump cued up on the tape in both cases. This will normally mean rewinding the tape after the first pass. You could record two dumps of Mult on the tape, following each other, and use one for the first pass and the other for the second pass.

Please study the above example carefully and try it out for yourself.

Disassembler & Monitor: SECTION 1

MONA3 is supplied in a relocatable form; you simply load it at the address that you wish it to execute from and then enter *MONA3* via that address. If you wish to enter *MONA3* again (having returned from *MONA3* to BASIC) then you should execute an address 2 (decimal) greater than the original address.

To load *MONA3* into your Amstrad CPC464, simply type:

`RUN "` **[ENTER]**

and press **[PLAY]** on the tape deck. Firstly, a short BASIC program will be loaded and this will automatically run itself and produce the message:

```
Load Address?
```

You should enter a decimal number between `1000` and `30000` and then press **[ENTER]**. The number you enter is the address at which the disassembler will be loaded - a good address is `30000`. After you have entered this number the message:

```
Please wait..loading MONA3.1..
```

will appear on the screen and the tape recorder will be started again automatically. The disassembler is now being loaded into the computer; this will take roughly 100 seconds. When *MONA3* has been loaded, it will run itself and produce a sign-on message on the screen - then the '*Front Panel*' will appear, ready for you to enter a command - see Section 2.

Note: if you intend to have both *GENA3*, the assembler, and *MONA3*, the disassembler/debugger, in the computer at the same time, then always load the one that is going lowest in memory last. It is normally a good idea to have *GENA3* in low memory (say at `1000`) and *MONA3* in high memory (say at `30000`). In this case *GENA3* should be loaded last.

Commands take effect immediately - there is no need to terminate them with **[ENTER]**. Invalid commands are ignored. The entire 'front panel' display is updated after each command is processed, so that you can observe any results of the particular command.

Many commands require the input of a hexadecimal number. When entering a hexadecimal number you may enter as many hexadecimal digits (0-9 and A-F or a-f) as you wish, and terminate them with any non-hex digit. If the terminator is a valid command, then the command is obeyed after any previous command has been processed. If the terminator is a minus sign '—', then the negative of the hexadecimal number entered is returned in two's complement form:

eg: `1800—` yields `E800`

If you enter more than four digits when typing a hexadecimal number, then only the last 4 typed are retained and displayed on the screen.

SECTION 2

The commands available.

The following commands are available from within *MONA3*. In this section, whenever **[ENTER]** is used to terminate a hexadecimal number this in fact can be any non-hex character (see Section 1). Also '_' is used to denote a space where applicable.

[CTRL]X

Return to BASIC or to whatever program called *MONA3*

[CTRL] D

flip the number base in which addresses are displayed between base 16 (hexadecimal) and base 1Ø (denary). On entry to *MONA3*, addresses are shown in hexadecimal, use **[CTRL] D** to flip to a decimal display and **[CTRL] D** again to revert to the hexadecimal format. This affects all addresses displayed by *MONA3* including those generated by the dis-assembler but it does not change the display of memory contents - 8 bit numbers are always displayed in hexadecimal, and numbers are always entered from the keyboard in hexadecimal.

[CTRL] A

Display a page of dis-assembly starting from the address held in the Memory Pointer. Useful to look ahead of your current position to see what instructions are coming up. Hit **[CTRL] A** again to return to the 'Front Panel' display or another key to get a further page of dis-assembly or **[ENTER]**.

'→' Cursor right

increment the Memory Pointer by one so that the 32 byte memory display is now centred around an address one greater than it was previously.

'←' Cursor left

decrement the Memory Pointer by one.

'↑' Cursor up

decrement the Memory Pointer by eight - used to step backwards quickly.

'↓' Cursor down

increment the Memory Pointer by eight - used to step forwards quickly.

'G'

search memory for a specified string ('**G**'et a string).

You are prompted with a '`:`' and you should then enter the first byte for which you want to search followed by **[ENTER]** - now keep entering subsequent bytes (and **[ENTER]**) in response to the '`:`' until you have defined the whole string. Then just press **[ENTER]** in response to the '`:`'; this will terminate the definition of the string and search memory, starting from the current Memory Pointer address, for the first occurrence of the specified string.

When the string is found the 'front panel' display will be updated so that the Memory Pointer is positioned at the first character of the string. Example:

Say that you wish to search memory, starting from `#8ØØØ`, for occurrences of the pattern `#3E #FF` (2 bytes) - proceed as follows;

`M:8ØØØ` **[ENTER]**	set the Memory Pointer to `#8ØØØ`.
`G:3E` **[ENTER]**	define the first byte of the string.
`FF` **[ENTER]**	define the second byte of the string.
[ENTER]	terminate the string.

After the final **[ENTER]** (or any non-hex character) '**G**' proceeds to search memory from `#8ØØØ` for the first occurrence of `#3E #FF`.

When found, the display is updated; to find subsequent occurrences of the string use the '**N**' command.

'H'

convert a decimal number to its hexadecimal equivalent.

You are prompted with '`:`' to enter a decimal number terminated by any non-digit (i.e. any character other than `Ø..9` inclusive). Once the number has been terminated, an '=' sign is displayed on the same line followed by the hexadecimal equivalent of the decimal number.

Now hit any key to return to the command mode. Example:

`H:41472_=A2ØØ`	here a space was used as the terminator.

'I'

intelligent copy.

This is used to copy a block of memory from one location to another - it is intelligent in that the block of memory may be copied to locations where it would overlap its previous locations.

'I' prompts for the inclusive start and end addresses of the block to be copied ('`First:`', '`Last:`') and then for the address to which the block is to be moved ('`To:`'); enter hexadecimal numbers in response to each of these prompts. If the start address is greater than the end address then the command is aborted -otherwise the block is moved as directed.

'J'

execute code from a specified address.

This command prompts, via '`:`', for a hexadecimal number - once this is entered the internal stack is reset, the screen cleared and execution transferred to the specified address. If you wish to return to the 'front panel' after executing code then set a breakpoint (see the '!' command) at the point where you wish to return to the display.

Example:

`J:BØØØ` **[ENTER]** executes the code starting at `#BØØØ`.

You may abort this command before you terminate the address by using **[ESC]**.

[CTRL] C

Continue execution from the address currently held in the Program Counter (PC).

This command will probably be used most frequently in conjunction with the '!' command - an example should help to clarify this usage:

Say you are single-stepping (using **[CTRL] S**) through the code given below and you have reached address `#892Ø`. You are now not interested in stepping through the subroutine at `#9ØØØ` but wish to see how the flags are set up after the call to the subroutine at `#88ØØ`.

```
891E 3EFF                LD   A,-1
892Ø CDØØ9Ø              CALL #9ØØØ
8923 2AØØ8Ø              LD   HL,(#8ØØØ)
8926 7E                  LD   A,(HL)
8927 111488              LD   DE,#8814
892A CDØØ88              CALL #88ØØ
892D 2ØØ3                JR   NZ,lab1
892F 32Ø28Ø              LD   (#8ØØ2),A
8932 211488 lab1:        LD   HL,#8814
```

Proceed as follows: set a breakpoint, using '!', at location `#892D` (remember to use '**M**' first to set the Memory Pointer) and then issue a '**[CTRL] C**' command.

Execution continues from the address held in the PC which, in this case, is #892Ø. Execution will then continue until the address at which the breakpoint was set (#892D) at which point the display will pause, now hit any key and the front panel will appear, and you can inspect the state of the flags etc. after the call to the subroutine at #88ØØ. Then you can resume single-stepping through the code.

'L'

tabulate, or list, a block of memory starting from the address currently held in the Memory Pointer.

'L' clears the screen and displays the hexadecimal representation and ASCII equivalents of the 160 bytes of memory starting from the current value of the Memory Pointer. Addresses will be shown in either hexadecimal or decimal depending on the current state of the Front Panel (see **[CTRL] D** above).

The display consists of 20 rows with 8 bytes per row, the ASCII being shown at the end of each row. For the purposes of the ASCII display any values above 127 are decremented by 128 and any values between 0 and 31 inclusive are shown as '.'.

At the end of a page of the list you have the option of returning to the main 'front panel' display by pressing **[ESC]**ape or continuing with the next page of 160 bytes by pressing any other key.

'M'

set the Memory Pointer to a specified address.

You are prompted with ':' to enter a hexadecimal address (see Section 1). The Memory Pointer is then updated with the address entered and the memory display of the front panel changes accordingly.

'M' is essential as a prelude to entering code, tabulating memory etc.

'N'

find the next occurrence of the hex string last specified by the **'G'** command.

'N' begins searching from the Memory Pointer and updates the memory display when the next occurrence of the string is found.

'O'

go to the destination of a relative displacement.

The command takes the byte currently addressed by the Memory Pointer, treats it as a relative displacement and updates the memory display accordingly.

Example:

Say the Memory Pointer is set to `#6800` and that the contents of locations `#67FF` and `#6800` are `#20` and `#16` respectively - this could be intepreted as a JR `NZ,$+24` instruction. To find out where this branch would go on a Non-Zero condition simply press **'O'** when the Memory Pointer is addressing the displacement byte `#16`. The display will then update to centre around `#6817`, the required destination of the branch.

Remember that relative displacements of greater then `#7F (127)` are treated as negative by the Z80 processor; **'O'** takes this into account.

See also the **'U'** command in connection with **'O'**.

'P'

fill memory between specified limits with a specified byte.

'P' prompts for `'First:'`, `'Last:'` and `'With:'`. Enter hexadecimal numbers in response to these prompts; respectively, the start and end addresses (inclusive) of the block that you wish to fill and the byte with which you want to fill the block of memory. Example:

```
P
First:7000 [ENTER]
Last:77FF [ENTER]
With:55 [ENTER]
```

will fill locations `#7000` to `#77FF` (inclusive) with the byte `#55` ('U').

If the start address is greater than the end address then **'P'** will be aborted.

'R'

Reads an object code file from tape - this file may have been produced by *MONA3*'s **'W'** command or *GENA3*'s **'O'** or **'*T'** commands. You are prompted to enter a filename (just press **[ENTER]** if you don't know the filename) and then the address at which you want the code to be loaded - you MUST specify this address.

'>'

set a breakpoint after the current instruction and continue execution.
Example:

```
9000 B7                 OR   A
9001 C20098             CALL NZ,#9800
9004 010000             LD   BC,0
9800 21FFFF             LD   HL,-1
```

You are single-stepping the above code and have reached `#9001` with a non-zero value in register A, thus the Zero flag will be in a `NZ` state after the `OR A` instruction. If you now use **[CTRL] S'** to continue single-stepping then execution will continue at address `#9800`, the address of the subroutine. If you do not wish to single-step through this routine then issue the '>' command when at address `#9001` and the `CALL` will be obeyed automatically and execution stopped at address `#9004` for you to continue single-stepping.

Remember, '>' sets a breakpoint after the current instruction and then issues a **[CTRL] C** command.

See the '**[CTRL] S**' command for an extended example of single-stepping.

'S'

update the Memory Pointer so that it contains the address currently on the stack (indicated by SP). This is useful when you want to look around the return address of a called routine etc.

'T'

dis-assemble a block of code, optionally to the printer.

You are first prompted to enter the '`First:`' and '`Last:`' addresses of the code that you wish to dis-assemble - enter these in hexadecimal as detailed in Section 1. If the start address is greater than the end address then the command is aborted. After entering these addresses you will be prompted with '`Printer?`'; answer '**Y**' to direct the dis-assembly to your Printer stream or any other value to send output to the screen.

Now you are prompted with '`Text:`' to enter, in hexadecimal, the start address of any textfile that you wish the dis-assembler to produce. If you do not want a textfile to be generated then simply press **[ENTER]** after this prompt. If you specify an address then a textfile of the dis-assembly will be produced, starting at that address, in a form suitable for use by *GENA3*.

If you want to use a textfile with *GENA3* then you must either generate it at, or move it to, the first address given by the assembler editor's '**X**' command because this is the address of the start of the text expected by *GENA3*. You must also tell *GENA3* where the end of the textfile is; do this by taking the 'End of text' address given by the dis-assembler (see below) and patching it into the TEXTEND location of *GENA3* - see the *GENA3* manual, Section 3.2. Then you must enter *GENA3* by the warm start entry point, to preserve the text.

If, at any stage when you are generating a textfile, the text would overwrite *MONA3* then the dis-assembly is aborted - press any key to return to the Front Panel.

If you specified a textfile address or disassembly to the printer, you are now asked to specify a 'Workspace:' address - this should be the start of a spare area of memory which is used as a primitive symbol table for any labels generated by the dis-assembler. The amount of memory needed is 2 bytes for each label generated. You cannot default this address.

After this, you are asked repeatedly for the '`First:`' and '`Last:`' (inclusive) addresses of any data areas that exist within the block that you wish to dis-assemble. Data areas are areas of, say, text that you do not wish to be interpreted as Z80 instructions - instead these data areas cause DEFB assembler directives to be generated by the dis-assembler.

If the value of the data byte is between `32` and `127` (`#20` and `#7F`) inclusive then the ASCII interpretaion of the byte is given e.g. `#41` is changed to 'A' after a `DEFB`. When you have finished specifying data areas, or if you do not wish to specify any, simply type **[ENTER]** in response to both prompts. The 'T' command uses an area at the end of *MONA3* to store the data area address and so you may set as many data areas as there is memory available; each data area requires 4 bytes of storage. Note that using 'T' destroys any breakpoints that were previously set - see the '!' command.

The screen will now be cleared. If you asked for a textfile to be created then there will be a short delay (depending on how large a section of memory you wish to dis-assemble) while the symbol table is constructed. This having been done, the dis-assembly listing will appear on the screen or printer - you may pause the listing at the end of a line by hitting any key, subsequently hit **[ESC]** to return to the Front Panel display or any other key to continue the dis-assembly. If an invalid opcode is encountered then it is dis-assembled as NOP and flagged with an asterisk '*' after the opcode in the listing.

At the end of the dis-assembly the display will pause and, if you have asked for a textfile to be produced, the message '`End of text` xxxxx' will be displayed; xxxxx is the address (in hexadecimal or decimal) that should be `POKE`d (low order byte first) into the *GENA3* location `TEXTEND` in order that the assembler can pick up this dis-assembled textfile on a warm start. When the dis-assembly has finished, press any key to return to the Front Panel display.

Labels are generated, where relevant (e.g. in `C30078`), in the form `LXXXX` where '`XXXX`' is the absolute hex address of the label, but only if the address concerned is within the limits of the dis-assembly. If the address lies outside this range then a label is not generated, simply the hexadecimal or decimal address is given. For example, if we were dis-assembling between `#7000` and `#8000`, then the instruction

```
C30078
```

would be dis-assembled as

```
JP   L7800
```

on the other hand, if we were dis-assembling between `#9000` and `#9800` then the `C30078` instruction would be dis-assembled as

```
JP  #7800 or JP  30720
```

if a decimal display is being used. If a particular address has been referenced in an instruction within the dis-assembly then its label will appear in the label field (before the mnemonic) of the dis-assembly of the instruction at that address but only if the listing is directed to a textfile.

Example:

```
T
First:8B [ENTER]
Last:9E [ENTER]
Printer? Y
Text: [ENTER]
First:95 [ENTER]
Last:9E [ENTER]
First: [ENTER]
Last: [ENTER]
008B FE16                CP   #16
008D 3801                JR   C,L0090
008F 23                  INC  HL
0090 37     L0090        SCF
0091 225D5C              LD   (#5C5D),HL
0094 C9                  RET
0095 BF524E              DEFB #BF,"R","N"
0098 C4494E              DEFB #C4,"I","N"
009B 4B4559              DEFB "K","E","Y"
009E A4                  DEFB #A4
```

'U'

used in conjunction with the **'O'** command.

Remember that **'O'** updates the memory display according to a relative displacement i.e. it shows the effect of a `JR` or `DJNZ` instruction.

'U' is used to update the memory display back to where the last **'O'** was issued.

Example:

```
 7200 47          71F3 77
 7201 20          71F4 C9
>7202 F2<        >71F5 F5<
 7203 06          71F6 C5
 display 1        display 2
```

You are on display 1 and wish to know where the relative jump `20 F2` branches. So you press **'O'** and the memory display updates to display 2.

Now you investigate the code following `#71F5` for a while and then wish to return to the code following the original relative jump in order to see what happens if the zero flag is set. So press **'U'** and the memory display will return to display 1.

Note that you can only use **'U'** to return to the last occurrence of the **'O'** command, all previous uses of **'O'** are lost.

'V'

used in conjunction with the **'X'** command.

'V' is similar to the **'U'** command in effect except that it updates the memory display to where it was before the last **'X'** command was issued. Example:

```
 8702 AF           842D 18
 8703 CD           842E A2
>8704 2F<         >842F E5<
 8705 84           8430 21
display 1         display 2
```

You are on display 1 and wish to look at the subroutine at `#842F`. So you press **'X'** with the display centred as shown; the memory display then updates to display 2. You look at this routine for a while and then wish to return to the code after the original call to the subroutine. So press **'V'** and display 1 will reappear.

As with **'U'** you can use this command only to reach the address at which the last **'X'** command was issued, all previous addresses at which **'X'** was used are lost.

'W'

Writes a block of memory to tape under a given filename. Prompts for a filename and then the first and last (inclusive) addresses of the block of code that you wish to save.

'!'

sets a *breakpoint* at the Memory Pointer.

A 'breakpoint', as far as *MONA3* is concerned, is simply a `CALL` instruction into a routine within *MONA3* that displays the Front Panel thus enabling the programmer to halt the execution of a program and inspect the Z80 registers, flags and any relevant memory locations. Thus, if you wish to halt the execution of a program at `#9876`, say, then use the **'M'** command to set the Memory Pointer to `#9876` and then use '!' to set a breakpoint at that address. The 3 bytes of code that were originally at `#9876` are saved and then replaced with a `CALL` instruction that halts the execution when obeyed. When this `CALL` instruction is reached it causes the original 3 bytes to be replaced at `#9876` and execution is paused - hit any key to enable the Front Panel to be displayed with all the registers and flags in the state they were just before the breakpoint was executed. You can now use any of the facilities of *MONA3* in the usual way.

Notes:

MONA3 uses the area, at the end of itself, that originally contained the relocation addresses in order to store breakpoint information. This means that you may set as many breakpoints as there is memory available; each breakpoint requires 5 bytes of storage. When a breakpoint is executed *MONA3* will automatically restore the memory contents that existed prior to the setting of that breakpoint.

Note that, since the **'T'** command also uses this area, all breakpoints are lost when the 'T' command is used.

Breakpoints can only be set in RAM. Since a breakpoint consists of a 3 byte CALL instruction a certain amount of care must be exercised in certain exceptional cases e.g. consider the code:

```
 8000 3E         8008 00
 8001 01         8009 00
 8002 18         800A 06
 8003 06         800B 02
>8004 AF<        800C 18
 8005 0E         800D F7
 8006 FF         800E 06
 8007 01         800F 44
```

If you set a breakpoint at #8004 and then begin execution of the code from location #8000 then register A will be loaded with the value 1, execution transferred to #800A, register B loaded with the value 2 and execution transferred to location #8005. But #8005 has been overwritten with the low byte of the breakpoint call and thus we now have corrupted code and unpredictable results will ocuur. This type of situation is rather unusual but you must attempt to guard against it - in this case single-stepping the code would provide the answer; see the **'[CTRL] S'** command below for a detailed example of single-stepping.

Also note that, when a breakpoint is detected, execution pauses, waiting for you to hit any key, before the front panel is displayed.

'X'
is used to update the Memory Pointer with the destination of an absolute CALL or JP instruction.

'X' takes the the 16 bit address specified by the byte at the Memory Pointer and the byte at the Memory Pointer +1 and then updates the memory display so that it is centred around that address. Remember that the low order half of the address is specified by the first byte and the high order half of the address is given by the second byte - Intel format. Example:

Say you wish to look at the routine that the code CDØ563 calls; set the Memory Pointer (using '**M**') so that it addresses the Ø5 within the CALL instruction and then press '**X**'. The memory display will be updated so that it is centred around location #63Ø5.

See also the '**V**' command in connection with '**X**'.

'**Y**'

enter ASCII from the Memory Pointer.

'**Y**' gives you a new line on which you can enter ASCII characters directly from the keyboard. These characters are echoed and their hexadecimal equivalents are entered into memory starting from the current value of the Memory Pointer. The string of characters should be terminated by **[ESC]** and **[DEL]** may be used to delete characters from the string. When you have finished entering the ASCII characters then the display is updated so that the Memory Pointer is positioned just after the end of the string as it was entered into memory.

[CTRL] S

single-step.

Prior to the use of '**[CTRL] S**' (or '**>**') both the Program Counter (PC) and the Memory Pointer must be set to the address of the instruction that you wish to execute.

'**[CTRL] S**' simply executes the current instruction and then updates the Front Panel to reflect the changes caused by the executed instruction.

Note that you can single-step anywhere in the memory map (RAM or ROM) but that you cannot single-step through EXX or EX AF,AF' instructions.

There now follows an extended example which should clarify the use of many of the debugging commands available within *MONA3* - you are urged to study it carefully and try it out for yourself.

Let us assume that we have the 3 sections of code shown on the next page in the machine, the first section is the main program which loads HL and DE with numbers and then calls a routine to multiply them together (the second section) with the result in HL and finally calls a routine twice to output the result of the multiplication to the screen (third section).

```
7Ø8Ø 2AØØ72          LD    HL,(#72ØØ) ;SECTION 1
7Ø83 ED5BØ272        LD    DE,(#72Ø2)
7Ø87 CDØØ71          CALL  Mult
7Ø8A 7C              LD    A,H
7Ø8B CD1D71          CALL  Aout
7Ø8E 7D              LD    A,L
7Ø8F CD1D71          CALL  Aout
7Ø92 21ØØØØ          LD    HL,Ø
```

```
7100 AF        Mult:     XOR   A       ;SECTION 2
7101 ED52                SBC   HL,DE
7103 19                  ADD   HL,DE
7104 3001                JR    NC,Mu1
7106 EB                  EX    DE,HL
7107 B2        Mu1:      OR    D
7108 37                  SCF
7109 C0                  RET   NZ
710A B3                  OR    E
710B 5A                  LD    E,D
710C 2007                JR    NZ,Mu4
710E EB                  EX    DE,HL
710F C9                  RET
7110 EB        Mu2:      EX    DE,HL
7111 19                  ADD   HL,DE
7112 EB                  EX    DE,HL
7116 30FB                JR    NC,Mu3
7118 B7                  OR    A
7119 20F5                JR    NZ,Mu2
711B 19                  ADD   HL,DE
711C C9                  RET
711D F5                  PUSH  AF      ; SECTION 3
711E 0F        Aout:     RRCA
711F 0F                  RRCA
7120 0F                  RRCA
7121 0F                  RRCA
7122 CD2671              CALL  Nibble
7125 F1                  POP   AF
7126 E60F      Nibble:   AND   %1111
7128 C690                ADD   A,#90
712A 27                  DAA
712B CE40                ADC   A,#40
712D 27                  DAA
712E CD5ABB              CALL  #BB5A
7131 C9                  RET
  .
  .
7200 1B2A                DEFW  10779
7202 0200                DEFW  2
```

Now we wish to investigate the above code either to see if it works or maybe how it works. We can do this with the following set of commands - it should be noted that this is merely one way of stepping through the code, it is not necessarily efficient but should serve to demonstrate single-stepping:

`M:7080` **[ENTER]**	set Memory Pointer to `#7080`.
`7080.`	set Program Counter to `#7080`.
[CTRL] S	single step.
[CTRL] S	single step.
[CTRL] S	follow the `CALL`
`M:7115`**[ENTER]**	skip the pre-processing of the numbers.
`!`	set a breakpoint.
[CTRL] C [ENTER]†	continue execution from `#7100` up to breakpoint.
[CTRL] S	single step.
[CTRL] S	follow the relative jump.
[CTRL] S	single step.
[CTRL] S	"
[CTRL] S	"
[CTRL] S	"
[CTRL] S	"
[CTRL] S	"
[CTRL] S	"
[CTRL] S	return from multiply routine.
[CTRL] S	single step
[CTRL] S	follow the `CALL`
`M:7128` **[ENTER]**	set Memory Pointer to interesting bit.
`!`	set breakpoint.
[CTRL]C [ENTER]†	continue execution from `#711D` to breakpoint.
[CTRL] S	single step.
[CTRL] S	"
[CTRL] S	"
[CTRL] S	"
`S`	have a look at the return address
`!`	set breakpoint there
[CTRL] C [ENTER]†	and continue.
[CTRL] S	single step.
`S`	return from `Aout` routine
`!`	
[CTRL] C [ENTER]†	
[CTRL] S	single step.
`>`	obey the whole `CALL` to `Aout`.

Please do work through the above example, first typing in the code of the routines (see 'Modifying Memory' below), or using *GENA3*, and then obeying the commands detailed above. You will find the example invaluable as an aid to understanding how to trace a path through a program.

† N.B. The **[ENTER]** at this position is an example of 'any key' needed to display the front panel.

[CTRL] L

this command is exactly the same as the '**L**'ist command except that the output goes to the Printer stream instead of to the screen. Remember that, at the end of a page, you press **[ESC]** to return to the 'front panel' or any other key (except **[CTRL] X**) to get another page.

Modifying Memory.

The contents of the address given by the Memory Pointer may be modified by entering a hexadecimal number followed by a terminator (see Section 1). The two least significant hex digits (if only one digit is entered then it is padded to the left with a zero) are entered into the location currently addressed by the Memory Pointer and then the command (if any) specified by the terminator is obeyed. If the terminator is not a valid command then it is ignored.

Examples:

`F2→`	`#F2`	is entered and the Memory Pointer advanced by 1.
`123 ↓`	`#23`	is entered and the Memory Pointer advanced by 8
`EM:EØØ_`	`#ØE`	is entered at the current Memory Pointer and then the Memory Pointer is updated to `#EØØ`. Notice that a space ('_') has been used to terminate the **M** command here.
`8CO`	`#8C`	is entered and then the Memory Pointer is updated (because of the use of the '**O**' command) to the destination of the relative offset #8C i.e. to its current value -115.
`2A5D_`	`#5D`	is entered and the Memory Pointer is not changed since the terminator is a space, not a command.

Modifying Registers.

If a hexadecimal number is entered in response to the '>' prompt and is terminated by a period, '.', then the number specified will be entered into the Z80 register currently addressed by the right arrow ' >'.

On entry to *MONA3* '>' points to the Program Counter (PC) and so using '.' as a terminator to a hex number initially will modify the Program Counter. Should you wish to modify any other register then use '.' by itself (not as a terminator) and the pointer '>' will cycle round the registers PC to AF. Note that it is not possible to address (and thus change) either the Stack Pointer (SP) or the IR registers.

Examples:

Assume that the register pointer '>' is initially addressing the PC.

.	point to IY.
.	point to IX.
Ø.	set IX to zero.
.	point to HL.
123.	set HL to #123.
.	point to DE.
.	point to BC.
E2A7.	set BC to #E2A7.
.	point to AF.
FFØØ.	set A to #FF and reset all the flags.
.	point to the PC.
8ØØØ.	set the PC to #8ØØØ.

[CTRL]J

This command enters the assembler GENA3, if this is present and has been used at least once. If the assembler is not present or has not been used, then **[CTRL]J** has no effect.